Distributed Mission Operations with the Multi-mission Encrypted Communication System[1]

Robert C. Steinke, Paul G. Backes, Jeffrey S. Norris
{Robert.C.Steinke, Paul.G.Backes, Jeffrey.S.Norris}@jpl.nasa.gov
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099

Abstract— The traditional model of mission operations is centralized with all activities taking place at a single location. The Multi-mission Encrypted Communication System (MECS) is a tool for enabling distributed operations where scientists and engineers at several locations collaborate over the Internet to perform mission operations activities. There are many reasons why distributed operations are desirable. Travel and facilities costs can be reduced. Disruption can also be reduced both at the mission operations facility which no longer has to house remote participants, and in the lives of remote participants who no longer have to leave their homes for weeks at a time. Finally, the level of participation can be increased leading to greater return from a mission.

The MECS architecture is centered around maintaining cached file replicas in a consistent state on remote machines. Challenges that are addressed by MECS include security, compatibility with legacy applications, clients that disconnect and reconnect to the network frequently, and user interface issues involved in keeping users informed when files are created or modified. This paper discusses MECS' architecture for distributed operations and lessons learned from a field test in May 2001.

TABLE OF CONTENTS

1. INTRODUCTION

Distributed mission operations is the concept of multiple participants at different physical locations collaborating to control a single spacecraft. Distributed operations are contrasted with the traditional centralized mission control room. Distributed operations have distinct advantages over centralized. Mission participants typically fall into two categories, engineers from the organization that launched the spacecraft, and scientists from universities and laboratories around the world. Centralized operations require that all the scientists converge on the engineers' facilities during mission operations.

This causes numerous problems. First, there is an enormous facilities burden on the mission operations site. The site normally supports only the engineers during spacecraft development, but must support engineers plus scientists during mission operations. The facilities will either be too small during operations, or mostly unused during development. There is also a financial burden of transporting scientists from their home institutions, and a personal burden on the scientists who must be away from their homes and families for weeks at a time. To relieve the personal burden scientists often work in shifts working for two weeks and then going home for two weeks. During the time at home they fall behind and need to catch up on what happened when they return. All of these problems can be ameliorated by distributed operations even in a limited form.

One result of these problems is to limit the number of people able to participate in a mission. Certain tasks such as data analysis are easily parallelized if enough people can have access to the data. With distributed operations, we imagine armies of graduate students sifting through data to find the needle in the haystack early enough for that information to be used to plan the spacecraft's next moves.

The concept of data distribution over the internet is not new, but until now it has not been applied to the application of mission operations. Remote participation in mission operations at JPL is also not new, but until now required installation of special hardware at the remote site called a Science Operations Planning Computer (SOPC) and leased data lines to JPL. Essentially, the remote computer was directly connected to the local area network of the mission operations site by a very long wire. This technique is very expensive which restricts its use to a few critical participants. It is also not mobile which we believe is one of the primary drivers for distributed operations.

The Multi-mission Encrypted Communication System (MECS) is being developed at JPL to support ubiquitous distributed operations on any computer that can run Java and connect to the Internet. This instantly includes a large number of mobile devices. MECS is also being designed to allow disconnected operation by caching mission database files on the user's machine. Currently, MECS is a work in progress. In this paper we distinguish between the requirements we placed on MECS, the design to fulfill those requirements, and we point out those parts of the design that we have not yet implemented.

2. MECS' Requirements

The primary goal of MECS is to enable distributed mission operations using only COTS hardware and the Internet as a communications medium. After initial security tasks to authorize a remote participant, adding a remote mission operations site should consist simply of downloading and installing the MECS software on any computer that the remote participant happens to be using.

MECS also needs to run with minimal supervision from system administrators. The result of these requirements is a nearly zero per-user marginal cost for promoting a centralized participant to a remote participant. Along with the travel and facilities cost savings, this should allow missions to increase their total number of participants which results in a greater number of person hours spent analyzing data during the critical time between downlink and the following uplink. A more detailed breakdown of some significant requirements is shown below:

1. Distributed read-write access to a mission database over the internet
2. Security
    a. Only authentic writes from authorized users shall be allowed to modify mission data
    b. Reads by authorized users shall return authentic mission data
    c. Reads by unauthorized users shall not be allowed to access mission data
    d. Easily revocable access control on a user by user, file by file granularity
3. Reliability
    a. Support for operations while disconnected from the Internet
    b. Previous mission database states recoverable
    c. Audit trail for writes
    d. Causal consistency enforced, concurrent writes detected
4. Efficiency
    a. Limit disk space on remote machine to specified allocation
    b. Scalable in number of users and size of mission database
    c. Load on remote machine should scale with number of files cached, and not number available to be cached
5. Usability
    a. Compatible with legacy applications manipulating mission data

The focus of MECS is to provide remote access to a mission database so that standard mission operations tools can be used in a distributed fashion. Initially, MECS is being designed to support primarily data analysis tools being used remotely. Data analysis is a read dominated workload and so is well suited for distributed operations. We also feel that other applications such as spacecraft commanding require greater security so acceptance of distributed operations will be slower.

Many current data analysis tools need only to access the mission database as a set of files. Their workload is read dominated although occasionally processed data needs to be written. Therefore, one of our initial decisions was that MECS should maintain a cached replica of the mission database on the remote user's file system. MECS monitors these files, and when one is changed by the user or an application program the new version is committed to the master database as soon as network connectivity allows. This supports compatibility with legacy applications and disconnected operations.

Disconnected operations are important because during centralized mission operations participants spend a significant amount of time traveling to and from the mission operations site. With disconnected operations they may spend this time analyzing mission data. We do not expect distributed operations to instantly replace centralized operations, and during the adoption period we expect mobile, disconnected operations to be one of the primary benefits of this new technology. For example, a scientist spends a few weeks at his or her home institution. During this time, the scientist's laptop keeps the mission database up to date any time it is plugged in to the internet. When the scientist travels back to the central mission operations site he or she can spend the travel time catching up, and be ready to go the instant he or she arrives.

An overriding concern with mission operations over the Internet is security. MECS addresses four security concerns. First, it must not be possible to impersonate a user and perform unauthorized modification of the mission database (requirement 2a.) Second, it must not be possible to impersonate the mission database and send to a user invalid data which the user believes to be authentic (requirement 2b.) Third, it must not be possible to impersonate a user or intercept data streams to perform unauthorized reads of the mission database (requirement 2c.) All three of these can be satisfied by implementing appropriate authentication, access control, and encryption. Fourth, access control must be fine grained and easily revocable in case a security breach is detected (requirement 2d.)

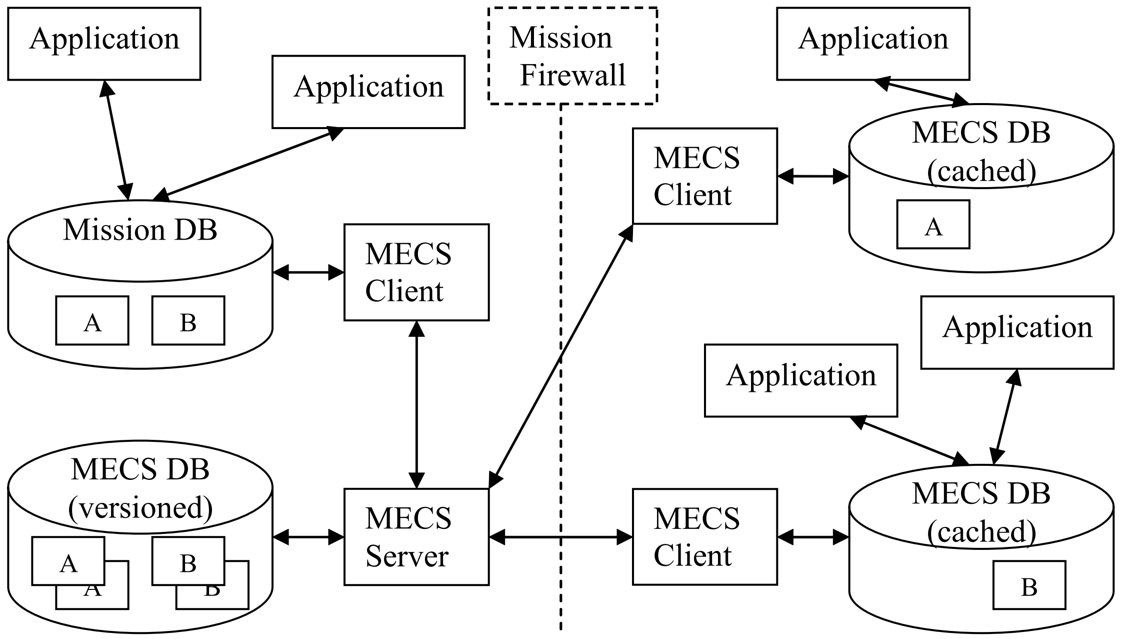


Figure 1 - MECS Architecture

3. MECS' Design

The architecture of MECS is based on a client-server architecture, and is shown in Figure 1. This architecture helps to fulfill several requirements. Security is fulfilled with NASA's Public Key Infrastructure[4] (NASA PKI) for authentication, triple DES EDE3[5] for encryption, and all access control decisions are made at the server behind the mission firewall after authentication. Having a centralized server makes access control easily revocable as opposed to, for example, a peer to peer protocol where access control decisions might be made by clients outside the mission firewall.

The server stores a copy of the mission database in version controlled form[2, 3]. This is represented by multiple copies of each file in the figure This fulfills the requirements of recoverability and auditability. It also provides support for disconnected operations and concurrent editing by remote participants as we shall show. Each client has a subscription specifying the set of files it is interested in caching. Not every user will be interested in the entire database, and subscriptions allow the user to limit disk usage on their local machine. A user can request a specific version of a file, subscribe to receive the newest version of a file, or subscribe to receive any new files created in a specific directory.

Behind the mission firewall, the original mission database remains and is monitored by a MECS client. Any new or modified files in the mission database are committed to the MECS server. This allows traditional, centralized mission operations to proceed exactly as before. MECS is not critical to the functioning of the system. MECS can even be switched off without affecting operations behind the firewall. In addition, MECS adds recoverability and auditability without any changes to existing mission operations tools. MECS can be configured to update files in the mission database when remote users commit changes or leave the mission database in its original state.

MECS uses three protocols for communication between clients and the server. All three use either TCP/IP or UDP and are implemented with Java sockets. The three protocols correspond to three actions that can take place within the system. The first protocol is the Subscription Protocol. For each client, the server stores a subscription specifying the files for which a client is interested in receiving updates. The Subscription Protocol allows a client to change its subscription. The second protocol is the Commit Protocol. When a file is created or modified in a directory monitored by a client that client commits the change to the server. The third protocol is the Update Protocol. When a change is committed to the server it must be propagated to all other subscribed clients. This is handled by the Update Protocol. Details of the three protocols are given below.

Subscription Protocol

In MECS, a subscription is defined as a set of files. The files in this set may or may not exist. For example, the specification "all of the files in directory foo" is taken to include an infinite number of files that don't exist, but could be created in the directory foo. A client with this subscription would receive an update if a file were created in foo. For each client, there are three important sets:

A	The set of files to which a client has read access
S	The set of files to which a client is subscribed
U	The set of files which have changed since the client was last updated

Files in the set $\neg A$ should never be sent to the client, and the set $\neg A \cap S$ should always be empty. The set $A \cap S \cap U$ consists of files that the client needs updated. Files in the set $A \cap \neg S$ are files the client may need if the client changes its subscription. The Subscription Protocol proceeds as follows:

1. The client opens a TCP connection to the server and sends the new subscription as a delta containing only changes from the old subscription.
2. The server applies the subscription change and sends the latest version numbers of all files in $A \cap \neg oldS \cap newS$. The client then requests those files for which it does not have the latest version.
3. If the server times out or receives invalid data the server aborts the protocol
4. If the client times out waiting for a response from the server it does not know whether its subscription was changed. Therefore, it must initiate a subscription repair protocol which is exactly like the subscription change protocol except that the client sends the entire subscription, not just a delta, and indicates that the server should use the empty set for oldS

In step 2, the server only sends version numbers because if the client is performing the subscription repair protocol the client will already have many of the files it needs.

Currently, we implement sets as simple lists of files and directories which are in the set. All files in a listed directory are assumed to be in the set. However, a more general form of these sets can be implemented efficiently as decision trees based on the directory hierarchy. The root of the tree specifies a file or directory included in the set. If it is a directory all of its descendants are assumed to be in the set, except that second level nodes specify descendants not included in the set. Third level nodes specify descendants of those descendants to include, and so forth. The status of a descendant not specifically mentioned is the status of its closest ancestor in the tree. A forest of these decision trees constitutes a set. With this implementation, a set containing all the descendents of a single directory is a one line specification, and we feel that the most often used sets will be very compact.

Scalability issues can be handled by having classes of users who all share the same subscription. For example, a public outreach program may have millions of users following along with a mission from their personal computers receiving data through MECS rather then from a web server. All of these users can be given the same subscription.

Commit Protocol

A commit creates a new version of a file in the server database. Any clients subscribed to that file will subsequently be updated with the new version via the Update Protocol. The Commit Protocol proceeds as follows:

1. The client opens a TCP connection to the server and sends a request for an Update Unique IDentifier (UUID)
2. The server responds with a UUID
3. The client sends the commit as its parent version in the version control graph and a delta containing only changes from that parent version. The client remembers the UUID in case the Commit Protocol fails.
4. The server applies the commit. If the parent version is the last version on its branch then the commit becomes a new revision on that branch. Otherwise, the commit becomes a new branch.
5. The server responds to the client that the commit succeeded, and includes the version number assigned to the commit.
6. If the server times out or receives invalid data the server aborts the protocol.
7. If the client times out waiting for a response from the server it does not know if the commit was applied. Therefore, the client must retry the commit protocol with the old UUID instead of requesting a new UUID. If the original commit succeeded the server will recognize the matching UUIDs and not apply the commit twice.

There are several important issues in implementing the commit protocol. We wish to transmit file deltas instead of entire files for efficiency. This means that it is important to enforce the condition that an update is applied at most once because the operation is not idempotent. It is also desirable to enforce the condition that the operation is applied at least once so that changes will not be lost. The given protocol does enforce at most once semantics, and if a client is diligent in retrying until the commit succeeds it enforces at least once semantics as well.

Another issue in transmitting deltas is the calculation of the deltas themselves. Directly comparing the file on the client with the file on the server would be as costly as transmitting the entire file from the client. An efficient algorithm using checksums to identify identical portions of two files over a network has been developed and integrated into the rsync program[1]. Of course, a brute force method exists by keeping two copies of each file on the client, one write protected, the other for the user to modify. A more efficient variation of this would be a file system with copy on write semantics where a single copy each disk block is kept until it is modified and then two copies are made. Many operating systems support copy on write semantics for memory, but we are unaware of any file system which supports it. We currently have not implemented transmitting deltas, and send entire files instead.

A third issue is enforcement of causal consistency. Since we allow disconnected operations it is entirely possible that two disconnected users could modify the same file in different ways at the same time. When these users reconnect, if one file were to overwrite the other then someone's work would be lost. This is an example of causally concurrent operations. When one performs a read then everything one does after that could have been caused by that read so the events are termed causally related. Two events that are not causally related are concurrent. It is invalid for one version to supercede a concurrent version, or lost updates can occur. We deal with this problem with the version control technique of branching. Two versions are causally related if and only if one is an ancestor of the other in the version control graph. There is also the problem of merging versions once a branch occurs. This is a very application specific problem because the correct actions will depend on what kind of data is stored in the file. We currently rely on system administrators to perform manual merging, but in the future we imagine plug-in modules specifying merging behaviors for particular file types.

There is a final problem with determining the parent version of a commit. MECS will always know the latest version it updated to the client, but the danger exists that an application program may bring the contents of a file into memory, hold them there while MECS updates a new version, and then write a modified old version of the file over the new version. This version should report the old version as its parent version, or causal consistency may not be enforced. We have not yet solved this problem. Solutions exist which require the cooperation of either users or applications, but this is less desirable because of our philosophy that MECS should be invisible to the user and compatible with legacy applications.

Update Protocol

When changes are committed to the server database, they must be propagated to subscribed clients. This is accomplished with the Update Protocol. The Update Protocol is an adaptation of a gossip protocol[6]. We chose this method because gossip protocols are very fault tolerant and reliable in the face of network disconnection, and we wish to support disconnected operation. A gossip protocol proceeds as follows. One host contacts another with information describing the messages it has received. The second host can deduce what messages it has received that the first host hasn't, and these messages are sent back to the first host. These pair-wise communications are repeated between random pairs of hosts until, eventually, all hosts receive all messages. Gossip protocols are traditionally peer to peer and require transmitting vector timestamps of size O(N), where N is the number of gossiping hosts, to indicate what messages a host has received. However, for security considerations, we have already disallowed peer to peer contact. One result of this is that all commits can be placed in a log at the server, and the log index, an O(1) timestamp, can be used to indicate what commits a client has received. The Update Protocol proceeds as follows:

1. Periodically, the client sends to the server a heartbeat message consisting of a single UDP packet containing the client's identity and the latest log index known to the client.
2. When the server receives a heartbeat message it checks the log index in the message against the size of the log. If records have been added to the log since the client's log index then the set U is non-empty and the server checks U against the client's subscription, S.
3. If U is empty the server does nothing as the client is up to date. The server can occasionally respond with a heartbeat message to the client so that the client can detect disconnection.
4. If U is non-empty then the server makes a TCP connection to the client and sends all the files in $U \cap S$, which may be empty, and the new highest log index.
5. In the event of any failure, both the server and client can abort and take no action. The client will eventually send another heartbeat with the same log index as before which is equivalent to retrying the protocol.

The first question we must address is why we use a polling algorithm when we could use a push based algorithm. After all, the server knows of both the commits and client subscriptions. Since we wish to deal with disconnected operations, we already need to detect disconnection which requires some form of heartbeat message. Since our polling information can fit in a single UDP packet it is no less efficient than disconnection detection. Furthermore, if the update protocol fails there must be some mechanism for remembering which updates have failed and retrying. We will have a single server and many clients so for scalability we wish this responsibility to fall on the clients. The protocol is also simplified by the fact that the first try and a retry require exactly the same actions.

There is another beneficial effect of this polling mechanism in the area of scalability. We expect most commits to come in large batches such as a downlink from a spacecraft. After this large commit there will be a period of peak server load followed by a period of very low server load. To increase the number of clients it is only necessary to reduce the frequency with which they send polling requests. The number of requests the server receives per unit time remains fixed while the duration of the period of peak load increases. Essentially, with fixed server peak load this algorithm supports a linear tradeoff between number of clients and average client latency in receiving updates.

4. Lessons Learned and Conclusion

From April 30th through May 11th, 2001 JPL conducted a blind field test of Mars rover mission operations to test several new technologies including MECS. The experimental Field Integrated Design and Operations (FIDO) rover[7] was placed at an unknown location which mission participants had to explore using only the rover's own capabilities. MECS was used to distribute data to a number of clients including Linux computers at the mission operations site that were not integrated with the local network file system, remote Sun workstations, and laptops running Microsoft Windows.

Twenty four days of operations were simulated over the twelve day field test, and MECS delivered all twenty four days of mission data without major incident. From this experience we feel confident in our claim that MECS will have a near zero per user cost. On the other hand, remote operations will make certain aspects of interpersonal communication more difficult.

To illustrate this point, a NASA administrator had expressed interest in seeing a demonstration of MECS, and had been given instructions before the field test on how to install and run MECS. During the test we became concerned when we had not heard from him thinking he had lost interest. We found out only later that he had been running MECS the whole time following along with the field test, and we were completely unaware of him.

So the first important lesson we learned is that with remote operations inter-personal communication will be a significant issue. Perhaps as significant as data communication. The ability to attract other people's attention in order to express an opinion should not be taken for granted. We are currently looking to groupware research for solutions.

Another issue is that while MECS can invisibly deliver the latest version of a file, there must be provisions for users to keep track of what files they have seen and what files are new. Often, measurements taken on the same day were downlinked on different days due to data volume constraints. There was no simple scheme for finding all of the new files. A primitive solution that we constructed during the field test was to display lists of files modified by each MECS update. This helped, but was not enough to be an acceptable solution.

Finally, we discovered the prevalence of firewalls in computer networks today. The last two days of the field test consisted of a public outreach demonstration involving several high schools. In order to use MECS to transmit data to these schools we had to deal with their firewall configuration and institutional policies for network security. We were not expecting to encounter this when dealing with a non-technical organization, but we certainly should have been.

References

[1] Andrew Tridgell and Paul Mackerras, "The rsync Algorithm", http://rsync.samba.org/rsync/tech_report.

[2] Marc J. Rochkind, "The Source Code Control System", IEEE *Transactions on Software Engineering*, Vol. SE-1, No. 4, December 1975, pp. 364-370.

[3] Walter F. Tichy, "RCS—A System for Version Control", *Software—Practice and Experience*, Vol. 15, No. 7, July 1985, pp. 637-654.

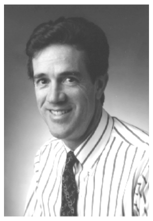
[4] NASA PKI home page, http://pki.nasa.gov.

[5] W. Tuchman, "Hellman Presents No Shortcut Solutions to DES", *IEEE Spectrum*, Vol. 16, No. 7, July 1979, pp. 40-41.

[6] Andrzej Pelc, "Fault-tolerant Broadcasting and Gossiping in communication Networks", *Networks*, Vol. 28, No. 3, 1996, pp. 143-156.

[7] FIDO home page, http://fido.jpl.nasa.gov.

Robert Steinke *is a computer scientist and member of the technical staff of the Autonomy and Control Section at the Jet Propulsion Laboratory. At JPL, his work is focused in the areas of distributed operations for Mars rovers and landers, secure data distribution, distributed data consistency, and communication support for group collaboration. He received his B.S. in 1995 and M.S in 1997 in Computer Science from U.C. Santa Barbara, and his Ph.D. in 2001 in Computer Science from the University of Colorado at Boulder. As a graduate student he completed his Master's thesis on a serializable lazy update scheme for distributed databases, and his dissertation on distributed shared memory consistency models, and received a graduate teacher certificate from Boulder for his training and work as an instructor. He now lives in Pasadena with his wife Mollie and daughter Adelaide.*



Paul Backes *is a technical group leader in the Autonomy and Control Section at the Jet Propulsion Laboratory, Pasadena, CA, where he has been since 1987. He received the BSME degree from U.C. Berkeley in 1982, and MSME in 1984 and Ph.D. in 1987 in Mechanical Engineering from Purdue University. He is currently responsible for distributed operations research for Mars lander and rover missions at JPL. Dr. Backes received the 1993 NASA Exceptional Engineering Achievement Medal for his contributions to space telerobotics (one of thirteen throughout NASA), 1993 Space Station Award of Merit, Best Paper Award at the 1994 World Automation Congress, 1995 JPL Technology and Applications Program Exceptional Service Award, 1998 JPL Award for Excellence and 1998 Sole Runner-up NASA Software of the Year Award. He has served as an Associate Editor of the IEEE Robotics and Automation Society Magazine.*

Jeff Norris *is a computer scientist and member of the technical staff of the Autonomy and Control Section at the Jet Propulsion Laboratory. At JPL, his work is focused in the areas of distributed operations for Mars rovers and landers, secure data distribution, and science data visualization. Jeff is a member of the ground data systems and operations teams for the 2003 Mars Exploration Rover Mission, and is contributing to the development of the rover command software suite. He received his Bachelor's and Master's degrees in Electrical Engineering and Computer Science from MIT. While an undergraduate, he worked at the MIT Media Laboratory on data visualization and media transport protocols. He completed his Master's thesis on face detection and recognition at the MIT Artificial Intelligence Laboratory. He now lives with his wife, Kamala, in La Crescenta, California.*